We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

1

Using Model Transformation Language Semantics for Aspects Composition

Samuel A. Ajila, Dorina Petriu and Pantanowitz Motshegwa
Department of Systems and Computer Engineering,
Carleton University, Ottawa, ON,
Canada

1. Introduction

Modern software systems are huge, complex, and greatly distributed. In order to design and model such systems, software architects are faced with the problem of cross-cutting concerns much earlier in the development process. At this level, cross-cutting concerns result in model elements that cross-cut the structural and behavioral views of the system. Research has shown that Aspect Oriented (AO) techniques can be applied to software design models. This can greatly help software architects and developers to isolate, reason, express, conceptualize, and work with cross-cutting concerns separately from the core functionality (Ajila et al., 2010; Petriu et al, 2007). This application of AO techniques much earlier in the development process has spawned a new field of study called Aspect-Oriented Modeling (AOM). In AOM, the aspect that encapsulates the cross-cutting behavior or structure is a model, just like the base system model it cross-cuts. A system been modeled has several views including structural and behavioral views. Therefore, a definition of an aspect depends on the view of interest. Unified Modeling Language (UML) provides different diagrams to describe the different views. Class, Object, Composite Structure, Component, Package, and Deployment diagrams can be used to represent the structural view of a system or aspect. On the other hand, Activity, State Machine, and Interaction diagrams are used to model the behavioral view. Interaction diagrams include Sequence, Interaction Overview, Communication, and Timing diagrams.

After reasoning and working with aspects in isolation, the aspect models eventually have to be combined with the base system model to produce an integrated system model. This merging of the aspect model with the base model is called Aspect Composition or Weaving. Several approaches have been proposed for aspect composition using different technologies/methodologies such as graph transformations (Wittle & Jayaraman, 2007), matching and merging of model elements (Fleury et al., 2007), weaving models (Didonet et al., 2006) and others. The goal of this research is to compose aspect models represented as UML sequence diagrams using transformation models written in Atlas Transformation Language (ATL).

Composing behavioral models (views) represented as UML Sequence diagrams is more complex than composing structural views. Not only is the relationships between the

model elements important but the order is equally paramount. Several approaches have been proposed for composing behavioral aspects with core system behavior, and these include graphs transformations [Whittle et al., 2007] and generic weavers [Morin et al., 2008]. In this research work we view composition as a form of model transformation. Aspect composition can be considered a model transformation since it transforms aspect and primary models to an integrated system model. Toward this end, we propose and focus on an approach that uses model transformations to compose both primary and aspect models represented as UML Sequence diagrams (SDs). SDs modeling the primary model and generic aspect models is created using Graphical UML modeling tools like Rational Software Architect (RSA). Model transformations are then used to instantiate the generic aspect models in the context of the application to produce context specific aspect models. Binding rules used for instantiating the generic aspect are represented as mark models that conform to a metamodel. Using other model transformations, the context specific aspect models are then composed with the primary model to produce an integrated system model. Verification and validation is performed, not only to verify that composition was successful, but also to ensure that the composed model is a valid UML model that can be processed further and shared with other researchers.

The rest of this chapter is structured as follows. Section two presents Model Driven approach, Aspect-Oriented techniques and technologies, and Atlas Transformation Language (ATL). We present our approach to model composition in section three starting with an example. We introduce our model composition semantics and definitions in section four – giving formal notions and three major algorithms (pointcut detection, advice composition, and complete composition) that define the basis of our work. Section five presents the design and implementation of our model composition using ATL semantics. We introduce and analyze a case study based on phone call features in section six. Section seven gives our conclusion, limitations, and future work.

2. Model Driven Engineering/Development/Architecture (MDE/MDD/MDA)

In Model Driven Engineering (MDE) everything is a model. A model refers to a simplified view of a real world system of interest; that is, an abstraction of a system. MDE considers models as the building blocks or first class entities (Didonet et al, 2006). A model conforms to a metamodel while a metamodel conforms to a metametamodel. MDE is mainly concerned with the evolution of models as a way of developing software by focusing on models. With this new paradigm of software development, the code will be generated automatically by model to code transformations. Model Driven Development (MDD) is copyrighted term by Object Management Group (OMG). One of the most important operations in MDE/MDD is model transformation. There are different kinds of model transformations including model to code, model to text, and model to model. Our interest in this paper is in model to model transformations. Figure 2.1 shows the process of model transformation. Since every artifact in MDE is a model, the model transformation is also a model that conforms to a metamodel. The transformation model defines how to generate models that conform to a particular metamodel from models that conform to another metamodel or the same metamodel. In Figure 2.1, the transformation model M_t transforms M_a to M_b. M_t conforms to MM_t while M_a and M_b conform to MM_a and MM_b respectively. The three metamodels conform to a common metametamodel MMM.

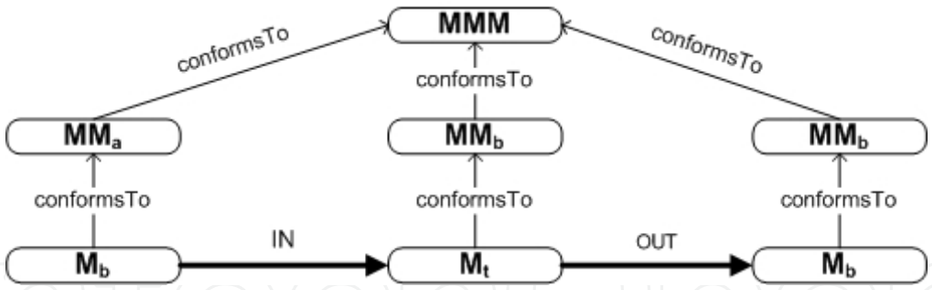


Fig. 2.1. Overview of Model Transformation Adopted from (ATL-User-Manual, 2009).

OMG's Model-Driven Architecture (MDA) is a term copyrighted by OMG that describes an MDE approach supported by OMG standards; namely, UML, Meta-Object Facility (MOF), MOF-Query/View/Transformation (QVT), XML Metadata Interchange (XMI) and Common Warehouse Metamodel (CWM). MDA decouples the business and application logic from the underlying platform technology through the use of the Platform Independent Model (PIM), Platform Specific Model (PSM) and model transformations. The PIM describes a software system independently of the platform that supports it while PSM expresses how the core application functionality is realized on a specific platform. Given a specific platform, the PIM is transformed to PSM. Platform in this case refers to technological and engineering details that are independent of the core functionality of the application. For example, middleware (e.g., CORBA), operating system (e.g., Linux), hardware, etc.

2.1 Aspect-Oriented (AO) techniques/technologies

The size of modern software systems has increased tremendously. Software architects and developers have to design and develop systems that are not only enormous, but are more complex, and greatly distributed. These systems naturally have many cross-cutting concerns (requirements) whose solutions tend to cross-cut the base architecture and system behavior. Such concerns include security, persistence, system logging, new features in software product lines, and many others. Aspect-Oriented techniques allow software developers and architects to conceptualize and work with multiple concerns separately (Groher & Voelter, 2007; Kienzle et al., 2009; Petriu et al., 2007). These techniques allow us to modularize concerns that we cannot easily modularize with current Object-Oriented (OO) techniques (Whittle & Jayaraman, 2007). The final system is then produced by weaving or composing solutions from separate concerns (Petriu et al., 2007). Klein et al. point out that dividing and conquering these cross-cutting concerns also allows us to better maintain and evolve software systems (Klein et al., 2007).

Aspect Oriented Programming (AOP) applies AO techniques at code level (France et al., 2004; Petriu et al., 2007). AOP was introduced to enhance Object-Oriented Programming to better handle cross-cutting concerns that cause code scattering and tangling, which leads to code that is very difficult to maintain and impossible to reuse or modify. AOP addresses these issues by introducing a class like programming construct called an *aspect* which is used to encapsulate cross-cutting concerns. Just like a class, an aspect has attributes (state) and methods (behavior). An aspect also introduces concepts well known to AO community; namely, *join points*, *advice*, and *pointcut*. Join points are points in the code where the cross-cutting behavior is to be inserted. AspectJ (a popular AOP Java tool) supports join points for

method invocations, initializing of attributes, exception handling, etc (Colyer et al., 2004). A *pointcut* is used to describe a condition that matches join points, that is it is a way of defining join points of interest where we want to insert the cross-cutting functionality. This cross-cutting behavior to be inserted at a join point is defined in the *advice*. AspectJ supports *before*, *after*, and *around* advices (Colyer et al., 2004).

Recent work in AO has focused on applying AO techniques much earlier in the development process (Kienzle et al., 2009; Klein et al., 2007; Morin et al., 2008; Petriu et al., 2007). In Aspect Oriented Modeling (AOM), Aspect-Oriented techniques are applied to models (unlike AOP). Whittle et al. define an AO model as "a model that cross-cuts other models *at the same level of abstraction*" (Whittle et al., 2006). Aspects are considered models as well; hence, it makes sense to define (or abstract) other concepts such as pointcuts and advices as models. However, the precise definition of a joint point, pointcut or advice depends on our modeling view. For example, in a structural view, such as a class diagram, an aspect is defined in terms of classes and operations/methods whereas in a behavioral view, such as a sequence diagram, an aspect is defined in terms of messages and lifelines. This has resulted in several approaches to AOM most of which have focused on separation and weaving (composition) of structural (class diagrams) and behavioral views (sequence, activity, and state diagrams) (Klein et al., 2007; Morin et al., 2008; Petriu et al., 2007). Several approaches have been proposed for composing aspects in AOM. These include using graph transformations (Gong, 2008; Whittle & Jayaraman, 2007), semantics (Klein et al., 2006), executable class diagrams (ECDs), weaving models (Didonet et al., 2006), generic approaches and frameworks (Fleury et al., 2008; Morin et al., 2008), etc. Composition primarily involves deciding what has to be composed, where to compose, and how to compose (Didonet et al., 2006). Aspect composition can either be symmetric or asymmetric. In symmetric composition, there is a clear distinction between the models to be composed; that is, one model plays the role of a base model while the other is declared an aspect model (Jeanneret et al, 2008). This distinction is absent in asymmetric composition.

2.2 Atlas transformation language

The Atlas Transformation Language (ATL) is a model transformation language from the ATLAS INRIA & LINA research group (ATL-User-Guide, 2009). The language is both declarative and imperative, and allows developers to transform a set of input models to a number of output target models. In ATL, source or input models can only be navigated but cannot be modified (Jouault & Kurtev, 2005) whereas target models are write-only and cannot be navigated. Figure 2.2 below shows an overview of an example of an ATL transformation (Family2Person) from [ATL_Examples] that transforms a Family model to a Person model. The Family model conforms to an *MMFamily* metamodel whereas the Person model conforms to an *MMPerson* metamodel. The ATL, *MMFamily*, and *MMPerson* metamodels all conform to the Ecore metametamodel which is a metamodel for the Eclipse Modeling Framework. Families2Persons.atl is an ATL model or program that transforms a Family model to a Person model.

ATL has three types of units that are defined on separate files (ATL-User-Guide, 2009); namely, ATL modules, queries and libraries. ATL has types and expressions that are based on the Object Constraint Language (OCL) from OMG. ATL has primitive types (Numeric, String, Boolean), collection type (sets, sequences and bags) and other types, all of which are

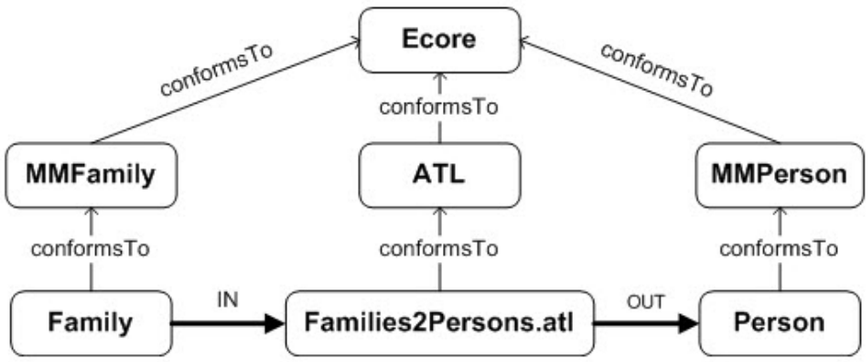


Fig. 2.2. Overview of the Family to Person ATL Transformation.

sub-types of the *OCLAny* abstract super-type. An ATL module or program, like Families2Persons.atl in the previous example, defines a model to model transformation (Jouault & Kurtev, 2005). It consists of a header, helpers (attribute and operation helpers) and transformation rules (matched, called and lazy rules) (Jouault & Kurtev, 2005). The header defines the module's name, and the input and target models. ATL operation helpers are more like functions or Java methods, and can be invoked from rules and other helpers. Attribute helpers unlike operation helpers do not take any arguments. All helpers are, however, recursive and must have a return value. Rules define how input models are transformed to target models. They are the core construct in ATL (Jouault & Kurtev, 2005). ATL supports both declarative and imperative rules. Declarative rules include matched rules and lazy rules. Lazy rules are similar to matched rules but can only be invoked from other rules. A matched rule consists of source pattern and target pattern (Jouault & Kurtev, 2005). A source pattern is defined as an OCL expression and defines what type of input elements will be matched (ATL-User-Guide, 2009). An ATL model is compiled, and then executed on the ATL engine that has two model handlers; namely, EMF (Eclipse Modeling framework) and MDR (Meta Data repository) (ATL-User-Guide, 2009). Model handlers provide a programming interface for developers to manipulate models (Jouault & Kurtev, 2005). The EMF handler allows for manipulation of Ecore models while MDR allows the ATL engine to handle models that conform to the MOF 1.4 (Meta Object Facility) metametamodel (ATL-User-Guide, 2009). For example, the ATL transformation in Figure 4.1 would require an EMF model handler since the metamodels conform to Ecore.

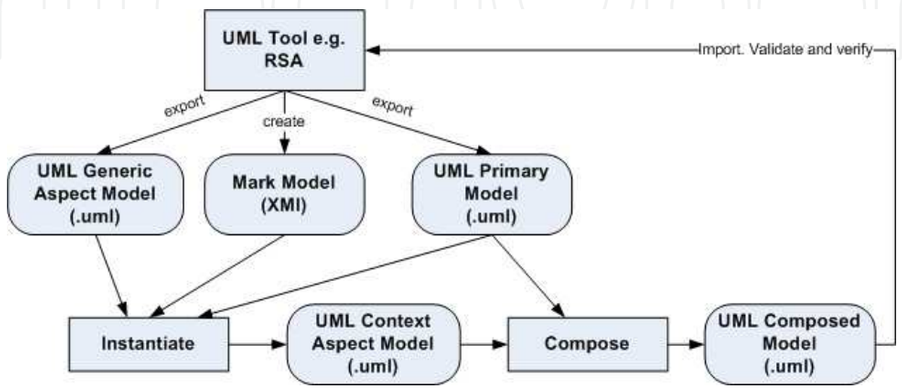


Fig. 3.1. Our AOM Composition Approach.

3. Our approach to model composition

Our approach shown in the Figure 3.1 is an adaptation of the approach proposed by Petriu et al. in (Petriu et al., 2007). Using a UML modeling tool, like RSA (Rational Software Architect) from IBM, the primary and generic aspect models are modeled in UML and then exported to a UML 2.1 (.uml) format file. The mark model is created in an XMI file. The *Instantiate* and *Compose* operations in Figure 3.1 are defined as ATL model transformations. We first instantiate a generic aspect model to the context of the application by using a model transformation that takes the primary, generic aspect and mark models as input, and transforms them to a context specific aspect model. We then invoke a second transformation that will take as input the newly created context specific aspect model and the primary model, and then output a composed target model.

3.1 Example

Let us introduce a simple example to provide a better view of our approach and the definitions of the various concepts used in the approach. This example is adapted from Klein et al. (Klein et al., 2007). It illustrates the weaving of a simple security aspect into a primary model. The primary model consists of a single scenario. In fact, our composition approach assumes that the primary model has only one sequence diagram (SD) and models a particular instance of a use case. This example consists of a login scenario shown in Figure 3.2 below. The model shows a simple iteration between instances of Server and Customer. The Customer attempts to log into the Server by sending a *login* message. The Customer's login details are incorrect; hence, the Server sends a *try_again* message to the Customer to attempt another login. The Customer then sends a *login* message with the correct details this time and the Server responds with an *ok* message.

The primary model does not have any security features. So, we want to add some security mechanism to the scenario so that when a customer attempts login and fails, the system should do something about that exception. We can model this security mechanism as a Security Aspect model that will detect a presence of a message from the Customer to the Server, and a reply from the Server back to the Customer. The presence of this sequence of messages is defined in the aspect's pointcut. The new behavior we want to add to the primary model in order to enhance security is defined in the aspect's advice. However, to make the aspect reusable and more useful, it has to be generic but not specific to our example or situation. This way we can reuse the aspect and in different situations and scenarios.

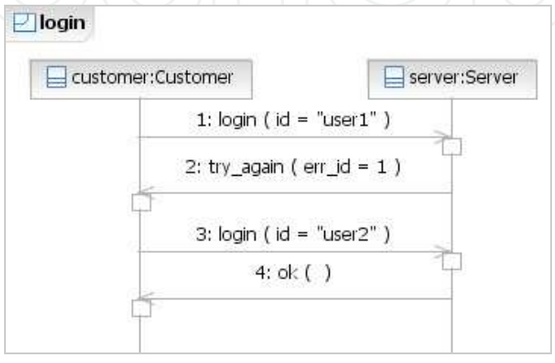


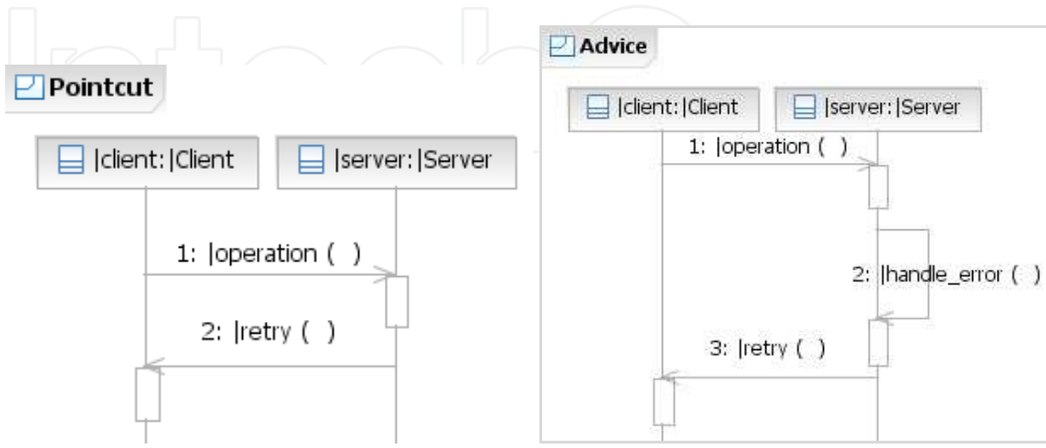
Fig. 3.2. The Primary Model - A Login Scenario for a Security Aspect Example.

To create a generic aspect we adopt the use of template parameters used by France et al. (France et al., 2004) and others (Kienzle et al., 2009; Petriu et al., 2007) to define generic roles played by the participants and messages in the generic aspect model. These generic roles are then bound to specific roles (names) when the generic aspect is instantiated. Figure 3.3 shows the pointcut and advice that make up our generic security aspect model. It should be noted that in case of multiple aspects, each aspect will be modeled separately. The lifelines (participants) and messages in the model are made generic. The pointcut in Figure 3.3a defines that the aspect detects any sequence of messages between a lifeline that plays the role of |client and lifeline that plays the role of |server such that |client sends a message tied to the role |operation and |server responds with |retry. During instantiation these template parameters (roles) will be set (bound) to concrete names of appropriate lifelines and messages.

As already mentioned, the advice represents the new or additional behavior we want executed if the pointcut matches, that is, if we find the sequence of messages defined in the pointcut in our primary model. The advice in Figure 3.3b declares that we want |server to invoke a self call after receiving |operation and before sending |retry to |client. So our advice in this case adds new behavior (the |handle_error self call). The idea is that during composition, as we shall see later, we replace whatever was matched by pointcut with what is defined in the advice.

Before an aspect can be composed with the primary model, the generic aspect model must first be instantiated to the context of the application to produce a Context Specific Aspect Model. This is achieved by "binding" the template parameters to application specific values. For example, we want to bind "customer" to |client because in our primary model, customer plays the role of |client.

Instantiating our generic aspect model using the bindings in Table 1, we obtain the context specific aspect model shown in Figure 3.4. The pointcut from the context specific aspect will then match the sending of a *login* message from *customer* to *server* and a try*_again* message from *server* back to *customer*, which is what we want. Its advice declares that the *save_bad_attempt* self call will be added to the server hopefully for the server to do something useful and security related.

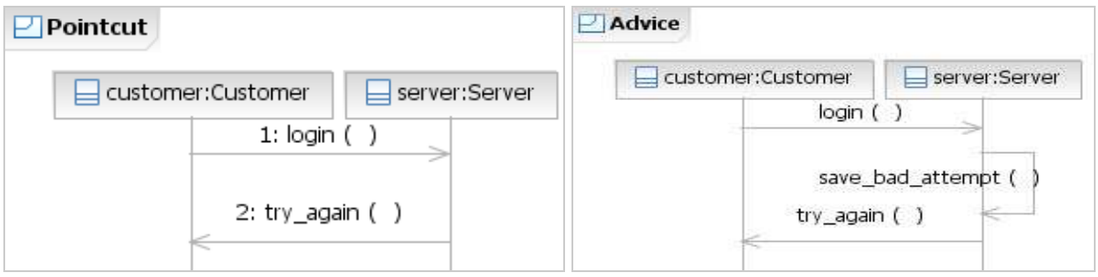


(a) pointcut (b) Advice

Fig. 3.3. Generic Aspect Model.

Parameter	Binding value	Comment
|client	customer	Lifeline object name.
|server	server	Lifeline object name.
|Client	Customer	The name of the type for the lifeline object.
|Server	Server	The name of the type for the lifeline object.
|operation	login	
|reply	try_again	
|handle_error	save_bad_attempt	

Table 1. Example of Security Aspect Binding Rules.



(a) Pointcut (b) Advice

Fig. 3.4. Context Specific Aspect Model.

3.2 Model composition

After instantiating a context specific aspect model, a complete integrated system is obtained by composing the primary model with the context specific aspect model. We view composition as a form of model transformation as shown in Figure 3.5. Therefore, our aim is to transform the input models (Primary and Context Specific Aspect) to a target composed model. As shown in Figure 3.5, both the input and output models conform to an EMF implementation of the UML metamodel specification while our ATL program or model conforms to the ATL metamodel. All the metamodels conform to the EMF's Ecore metametamodel. As in other aspect composition approaches composition has to be performed on different views, that is, structural or behavioral views. Our main focus is on the behavioral view. Composition inevitably results in some model elements been replaced, removed, added or merged (Morin et al., 2008; Gong, 2008). Similarly in our approach, all model elements from the context specific aspect model that are not already in the primary model, will be added to the composed model but elements common to both models will be merged. All join point model elements (from primary model) are replaced by advice elements. The rest of the elements from the primary model will be added to the composed model. A formal definition of our models and the proposed algorithm (for matching and composing) are based on UML metamodel classes. The specification for the UML metamodel (OMG, 2009) is enormous and also includes metaclasses for other UML diagrams that we are not interested in. Therefore, it makes sense to look only at some of the important classes whose objects are used in creating sequence diagrams (SDs).

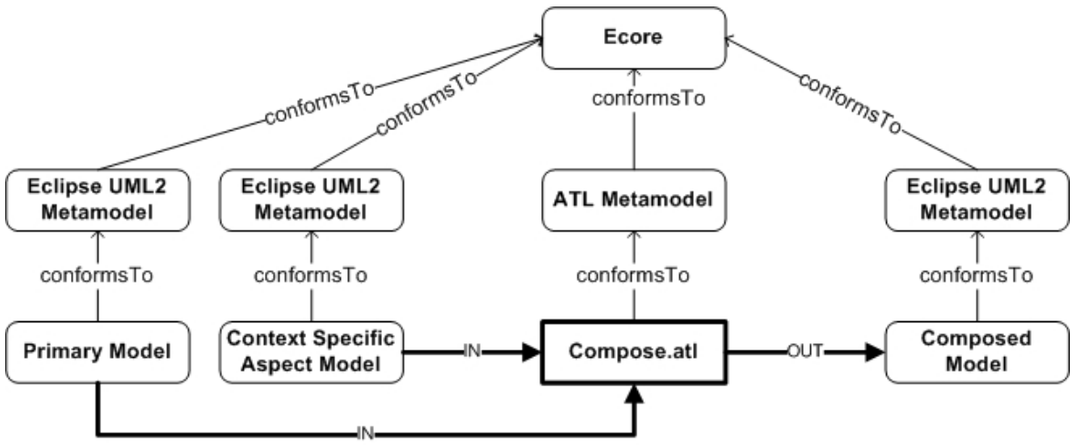


Fig. 3.5. Aspect Composition as an ATL model transformation.

4. Model composition semantics and definitions

A sequence diagram shows the order in which messages are exchanged among participants; hence, order is crucial [Hamilton & Miles, 2006; Pilone & Pitman, 2005). The messages or interactions, to be precise, on a specific lifeline are totally ordered but interactions between two lifelines are partially ordered. The most important model elements in a SD are probably lifelines (participants), messages, message ends, and the enclosing interaction. Figure 4.1 is a simplified class diagram of the Interactions Metamodel showing the relationships among the metaclasses for these model elements.

A complete description of each metaclass can be obtained from the UML specification (OMG, 2009). The *InteractionFragment* abstract class represents a general concept of an interaction (OMG, 2009). An *Interaction* is a sub class of *InteractionFragment* that represents the modeled behavior or interactions (exchange of messages) between participants (lifelines)[OMG09]. An *Interaction* essentially encloses *Messages*, *Lifelines* and other *InteractionFragments*. The enclosed *InteractionFragments* are stored in an ordered list referenced by the *fragment* role. This ordering is exploited in our algorithms for matching and composing SDs. A *Message* models the kind of communication between participants [OMG09]. There are five main types of messages; namely, synchronous, asynchronous, delete, create, and reply messages [Hamilton+06]. Each message is accompanied by a pair of MessageOccurrenceSpecifications (MOSs). The *sendEvent* MOS represents the sending of the message while *receiveEvent* MOS models the reception of the message. Each MOS also has a reference to the lifeline for which the message is received or sent from through the *covered* association. In return, each *Lifeline* has a reference to a list of *InteractionFragments* or specializations of *InteractionFragment* (including MOSs), which cover the lifeline, through the *coveredBy* association.

The events that we are interested in are specializations of the *MessageEvent* abstract class mainly the *SendOperationEvent* (SOE) and *ReceiveOperationEvent* (ROE) classes. These types of events occur during the sending or receiving of a request for an operation invocation (OMG, 2009).

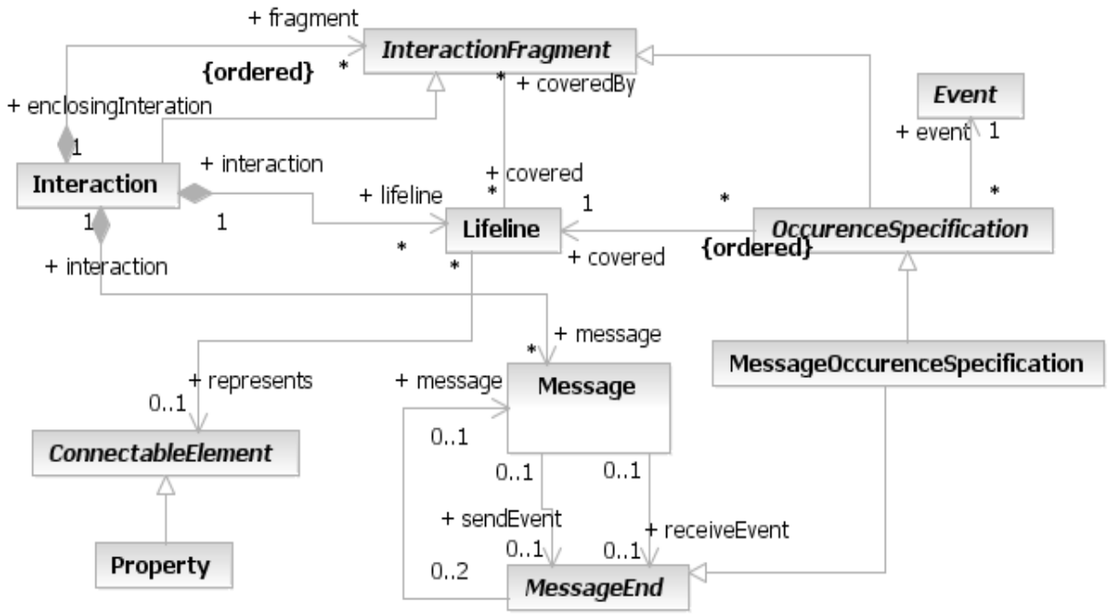


Fig. 4.1. Simplify Metamodel for Sequence Diagrams.

4.1 Sequence Diagram (SD) Composition

As previously described, our AOM approach has a primary model, one or more generic aspect models and a mark model. The primary model describes the core system functionality (behavior) without cross-cutting concerns. The generic aspect models describes (encapsulate) cross-cutting concerns which could otherwise be scattered across core functionality; for example, new features (in software product lines), security, persistence, etc. Before composing the primary model with an aspect model we first instantiate the generic aspect model in the context of the application with the help of a mark model. We employ an ATL transformation model that takes the primary, generic aspect, and mark models as input, and produces a context specific aspect model as output. We would like to point out that the mark model does not necessarily have to specify all the bindings for the template parameters in cases where some of the bindings can be matched or implied from the primary model. A second ATL transformation model then takes as input the primary and context specific models to produce the composed model. Defining a generic aspect improves re-usability since the same aspect can be instantiated and then composed with the primary model multiple times until a complete integrated system model is obtained. Since we are mainly interested in the behavioral view (of our primary and aspect models), our work is mainly focused on the composition of interactions diagrams in the form of SDs. As described earlier, the aspect model consists of a pointcut and an advice defined as SDs where the pointcut is the behavior to detect and the advice is the new behavior to compose or weave at the join points [Klein et al., 2007]. Before composing, we first have to identify all our join points by matching the pointcut SD with the primary model. The pointcut SD consists of message or a sequence of messages between lifelines; therefore, we want to find the occurrence of these sequences of messages in the primary model and then insert the defined cross-cutting behavior (defined in the advice SD) at every join point. Composition is essentially inserting this new behavior; that is, composition is achieved by replacing the join

points with the advice SDs. Before instantiating a generic aspect, we first have to ensure that the aspect can be applied to the primary model; that is, whether its pointcut matches. A formal definition of matching will be given later. Also during composition we have to find where to weave. This makes pointcut detection or finding join points a core operation. The algorithm designed for pointcut detection manipulates the SD metaclasses by exploiting the relationship between *InteractionFragments* and their ordered list of fragments in an interaction. It also makes use of the fact that a sequence of messages (and indeed a SD) is essentially a list of ordered fragments.

4.1.1 Formal notations for defining aspects and primary models

Let,

- **P** be a sequence of fragments, that is, *InteractionFragments* (CombinedFragments and MOSs), from the aspect's pointcut SD.
- **A** be a sequence of fragments from the aspect's Advice SD.
- **C** be a sequence of fragments from the primary model SD.

Note that a sequence is an ordered collection/list.

Then, **P** = Sequence{f_1, ..., f_ψ} where ψ = number of fragments in **P** and f_i is either a *CombinedFragment* (CF) or a MessageOccurrenceSpecification (MOS), such that,

f_i =	If instance of	where
CF(O, Λ)	CF	**O** is a sequence of operands in the **CF** and each operand is also a sequence of fragments just like **P**. This is the case with nested CFs. **Λ** is a list of lifelines covered by the **CF**.
MOS(L_i,E_i,M_i)	MOS	L_i is a lifeline covered by f_i. E_i is an event associated with f_i. M_i is a message associated with f_i.

and,

C = Sequence{c_1, ..., c_μ} where μ = number of fragments in **C** and c_i is also either a CF or a MOS, such that,

c_i =	If instance of	where
CF(O, Λ)	CF	**O** is a sequence of operands in the **CF** and each operand is also a sequence of fragments just like **C**. This is the case with nested CFs. **Λ** is a set of lifelines covered by the **CF**.
MOS(L_i,E_i,M_i)	MOS	L_i is a lifeline covered by c_i, E_i is an event associated with c_i. M_i is a message associated with c_i.

4.1.2 Aspect and primary models definition

Using the above notation, we will define an aspect model as a pair of fragment sequences, that is, **Aspect** = (**P**, **A**) where **P** and **A** are the sequences defined earlier. This definition is adapted from Klein et al. in (Klein et al., 2006; Klein et al., 2007); However, Klein et al. define

a simple SD as a tuple that consists of a set of lifelines, a set of events, a set of actions and partial ordering between the messages (Klein et al., 2007). This is different from our definition of a sequence of fragments. Using our definition, the primary model = **C**, a sequence of fragments from the primary model SD. Then our pointcut **P** matches **C** if and only if there exists two sub-sequences M_1 and M_2 in C such that, $C = M_1 \oplus P \oplus M_2$, where $\oplus$ denotes a union of sequences. $A \oplus B$ returns a sequence composed of all elements of A followed by the elements of B. If the **P** matches **C** several times, say *n* times, then we can say, $C = M_1 \oplus P \oplus M_2 \oplus \ldots \oplus M_n \oplus P \oplus M_{n+1}$. This definition is an adaptation of the definition given by Klein et al. in (Kleinet al., 2006).

4.1.3 Join point definition

Part of the sequence **C** that corresponds or matches **P** is the join point. In other words, a join point is a sub sequence of **C** that is equal to the sequence **P**. Equal here means that fragments at the same corresponding location in **P** and join point (same index on either sequences) are equal. For example, if elements at position 1 in **P** and in the join point are both MOSs, they can only be equal if and only if they cover similar lifelines (same name and type), have the same message, and have other features that are similar. More details for checking for equality will be given in the design and implementation section. Since the size (number of fragments) of **P**, hence the size of a join point, is fixed we can afford to keep track of only fragments at the beginning of each join point. With this assumption we can define; **S** = Sequence$\{s_1, \ldots, s_n\}$, a sequence of fragments at the beginning of each join point where $n > 1$ is number of join points matched by **P**. A join point, J_i is then given by a sub sequence of **C** from index of s_i in **S** to the index of s_i plus ψ minus 1. That is, if; x_i = **indexOf**(s_i) and $y_i = x_i + \psi - 1$, where ψ = number of elements in **P**, then, J_i = C->**subSequence**(x_i,y_i) for $1 \leq i \geq n$.

4.2 Composition algorithms - assumptions and requirements

The below algorithm and indeed the other algorithms to be introduced later, make the following assumptions:

- The input models are well formed and valid; hence, the sequences **S**, **P**, and **A** are valid. For example, we do not have empty sequences. We also assume that the aspect models (generic and context specific) consists of two interactions (SDs) named Advice and Pointcut, and that the primary model represents one interaction or scenario; therefore, consists of one instance of *Model*, one instance of *Collaboration*, and one instance of *Interaction*.
- We can correctly compare any two fragments regardless of their specialization, for example, comparing a MOS with a CF.
- Nested CFs have been properly and consistently unrolled.
- A lifeline's name is the same as that of the represented object (property).
- Message have arguments with primitive UML types (strings and integers).
- We can ignore other fragments like *BehaviorExecutionSpecifications* (BESs) and *ExecutionOccurrenceSpecifications* (EOSs) focusing only on MOSs and CFs (and their operands), and still achieve accurate pointcut detection.

4.2.1 Pointcut detection algorithm

The pseudo code for the algorithm that detects or matches pointcuts and returns **S** is given below. The algorithm begins by creating an empty sequence **S** on line 2. It then iterates over all fragments c_i in **C** checking if c_i is equal to f_1, the first element in our pointcut **P** on line 9. If the elements are **not** equal, the algorithm moves to the next c_i. However, if the fragments (c_i and f_1) are equal, it obtains $\mathbf{J_i}$, a sub sequence of **C** starting from c_i and with the same size as **P**, on line 10.

Algorithm-1 Pointcut Detection Algorithm
Input : **P** = Sequence{f_1, ..., f_ψ}, **C** = Sequence{c_1, ..., c_μ}
where ψ = number of fragments in **P**, and μ = number of fragments in **C**
Output : **S** = Sequence{s_1, ... ,s_n}

```
1.    begin
2.        S = Sequence{}
3.        foreach ci in C do
4.            //compute the location of the end of the potential join point
5.            k = i + ψ -1
6.            if k > μ then //make sure we have a valid location
7.                break
8.            end if
9.            if ci = f1 then
10.               Ji = C->subSequence(i,k) // Potential join point
11.               /* check if fragment at the same location in P is equal to the
12.               corresponding element in the join point */
13.               if pairWiseMatch(P, Ji) then
14.                   S->enqueue(ci)
15.               end if
16.           end if
17.       end loop
18.       return S
20.   end
```

On line 13 the algorithm then compares **P** and $\mathbf{J_i}$, side-by-side by checking if each pair of fragments at index *j* on both sequences is equal for $1 \leq j \geq \psi$. If this is true, then indeed $\mathbf{J_i}$ is a join point. So the algorithm inserts the first element (c_i) of the join point into **S** and loops back to line 3. It continues looping until it has checked all the elements of **C** or the condition on line 6 is true to ensure we do not fall off the edge. More details on the implementation of this algorithm and its functions, like *pairWiseMatch*, will be discussed in the next section.

4.2.1.1 Algorithm-1 complexity

If the algorithm-1 has to visit all fragments in **C** (when ψ = 1) then both functions on lines 10 and 13 will take constant time, that is, O(1) which makes the algorithm linear or O(n). If **P** is the same size as C (ψ = μ), then the algorithm has to loop only once but both *subSequence* and *pairWiseMatch* functions are O(n); hence, the algorithm is again linear. However, if ψ < μ then again both functions (i.e., *Sequence* and *pairWiseMatch*) are, in the worst case, linear and the algorithm will have to loop several times each time invoking the two functions making the algorithm quadratic, that is $O(n^2)$; therefore, in general the algorithm is $O(n^2)$.

4.2.2 Complete composition algorithm

After detecting the pointcut and obtaining our join points, the next step is to weave the advice at the join points. Since the advice has already been bound to the context of the application during aspect instantiation, weaving the advice is simplified to replacing a join point with the advice. This is trivial with only one join point. Challenges arise when we have multiple join points because we have only one advice from the aspect model. We can either duplicate the advice or work with one join point at a time. Both options were explored but duplicating the advice (without duplicating the aspect model) proved to be complex due to the inability to navigate target models in ATL, and the nested relationships between *InteractionFragments*. Focusing one join point at a time is easier and more elegant. The complete composition algorithm presented in this section achieves this. Let us first introduce abbreviations that we will use in the algorithm.

- **GAM** = Generic Aspect Model
- **CSAM** = Context Specific Aspect Model (Instantiated generic aspect model)
- **MM** = Mark Model
- **PM** = Primary Model
- **CM** = Composed Model

The pseudo code of the Complete Composition Algorithm is given below. The three functions defined in this algorithm represent the ATL transformations used to implement this algorithm as we shall see in the next chapter. The algorithm begins by retrieving *n* the number of join points matched in the primary model (PM) using the *JoinPointsCount* function on line 2. This function implements algorithm-1 (Pointcut detection Algorithm) to return a sequence of fragments at the beginning of each join point, and then finds the size of that sequence. The details of the implementation of this function will be given in next chapter. The number of join points determines if the algorithm will execute lines 6 to 10, and not necessarily the number of times the loop will iterate. If n > 0, that is, we have at least one join point, the algorithm instantiates the GAM to create a CSAM on line 6. This corresponds to the instantiate process shown in Figure 5.1. It then composes PM with CSAM by weaving the advice at the first join point using the *Compose* function on line 8 to produce our composed model.

Algorithm-2 Complete Composition Algorithm

```
Input     :GAM, MM, PM
Output    :CM
1.        begin
2.            n = JoinPointsCount(GAM, MM, PM)
3.            temp = PM
4.            while n > 0
5.                // instantiate our generic aspect model
6.            CSAM = Instantiate(GAM, MM, temp)
7.                // compose advice and first join point
8.                CM = Compose (temp,CSAM)
9.            temp = CM
10.           n = JoinPointsCount(GAM, MM,temp)
11.           end while
12.           return CM
13.       end
```

The algorithm then checks the CM for more join points on line 10. If more are found, it returns to line 6 to instantiate the GAM using CM (new primary model). It then creates another CM and checks for more join points. The algorithm continues looping until no join points are found.

As it is, this algorithm has a potential nasty flaw in the form of positive feedback, which, if left unattended, can cause the algorithm to loop indefinitely in some cases! The problem is rooted on the fact that composing an aspect in most cases results in the addition of new model elements (fragments, messages and lifelines) which in turn can produce more join points. This means that after composition on line 8, the algorithm may find more join points on line 10 causing the algorithm to iterate again and again. For example, if the pointcut is defined as a single message MSG1, and the primary model has two invocations of this message, then we have two join points. If the advice adds three instances of the same message MSG1, then after composition (1st iteration) we will have four join points. After the second iteration well have six, then eight, etc. With the number of join points increasing all the time the algorithm will never terminate. This problem is easily solved by tagging model elements from advice during instantiation on line 6. To be precise we only have to tag MOSs (fragments). Then when pointcut matching during the invocation of *JoinPointsCount* (implementing algorithm-1), we check for that tagging. If a potential join point has at least one tagged fragment, then we know that this join point emerged only after composition; therefore, it is immediately disqualified.

4.2.2.1 Algorithm-2 complexity

The complexity of algorithm-2 is difficult to analyze because on the surface the algorithm appears to be linear on the number of join points. However, the algorithm is not necessarily linear on the number fragments. We have already seen that detecting the number of join points is quadratic. Therefore, if that is nested within a loop, we could say that (in general) the algorithm is cubic, that is, $O(n^3)$

4.2.3 Advice composition algorithm

At the core of the *Compose* function, used by the Complete Composition Algorithm described above, is the Advice Composition algorithm that weaves the advice at the join point. Recall the definition of an **Aspect** = (**P**, **A**). We will use definition again where by "**Aspect**" we are referring to a context specific aspect model. Our main interest is mainly on the advice sequence **A**. Recall that,

- **A** = Sequence$\{a_1, ..., a_\omega\}$, a sequence of fragments from the aspect model advice SD, where ω = number of fragments in **A**.
- **C** = Sequence$\{c_1, ..., c_\mu\}$, a sequence of fragments from the primary model, where μ = number of fragments in **C**.
- **S** = Sequence$\{s_1, ... ,s_n\}$, a sequence of fragments at the beginning of each join point, where n > 1 is number of join points matched by **P**.
- A join point, $\mathbf{J_i}$ is then given by a sub sequence of C from index of s_i in **S** to the index of s_i + ψ-1; That is, If, x_i = indexOf(s_i) and $y_i = x_i + \psi - 1$, then, $\mathbf{J_i}$ = C->subSequence(x_i,y_i) for $0 \leq i \geq n$
- **P** is a sequence of fragments from the pointcut SD.

Since our Complete Composition Algorithm is concerned with one join point at a time, our Advice Composition algorithm needs to work with only one join point; that is, the join point that begins with s_1 (the first element in **S**). Then, let $\mathbf{C_{CM}}$ be a sequence of fragments from the composed model. Recall again that;

With the notation defined, we can now describe our Advice Composition algorithm. Its pseudo code is given on the next page. In a nut shell, the algorithm simply replaces the join point with the advice. The algorithm first checks if we have a join point. If so, it obtains the first element of **S**, on line 5. Using that element, the algorithm finds the location (x_1) at the beginning and at the end (y_1) of the join point, as shown on lines 6 and 7. The algorithm then obtains a sub sequence of fragments from **C** (primary model) before the start of the join point, on line 12. Note that indexing for our sequence data structure starts at 1 instead of zero as in Java lists or arrays. On line 16, the algorithm returns a sequence of fragments after the last element of the join point to the end of **C**. The composed model is then given by $\mathbf{C_{CM}}$ = *sub_before* ⊕ **A** ⊕ *sub_after*, that is, the union of *sub_before*, **A** and *sub_after*.

Algorithm-3 Advice Composition
Input : **C**, **A**, **P**, **S -** where ψ = number of fragments in **P**
Output : $\mathbf{C_{CM}}$

```
1.  begin
2.      if S->isEmpty() then
3.          C_CM = {}
4.      else
5.          s1 = S->first()        // fetch the tail of the 1st join point
6.          x1 = C->indexOf(s1)    // find its location in C
7.          y1 = x1 + ψ–1 // find the location of the join point's head
8.          sub_before = {}
9.          sub_after = {}
10.         if x1 > 1 then
11.             // get all fragments before the join point
12.             sub_before = C->subSequence(1,x1-1)
13.         end if
14.         if y1 < μ then
15.             // get all fragments after the join point
16.             sub_after = C->subSequence(y1+1, μ)
17.         end if
18.         //Insert the advice in place of the join point
19.         C_CM = Sequence {sub_before, A, sub_after}
20.     end if
21.     return C_CM
22. end
```

4.2.3.1 Algorithm-3 complexity

Creating *sub_before* and *sub_after* is linear in the worse case. Creating C_{CM} is also O(n) in the worst case; hence, the above algorithm is clearly linear.

5. Design and Implementation

In the previous section, we introduced our definition of primary, aspect and mark models. We also introduced our approach to AOM composition, and discussed our Complete Composition Algorithm that uses two other algorithms to detect join points, and compose the primary and aspect models. In this chapter we describe how the Complete Composition Algorithm was implemented using ATL transformations to realize the functions *JoinPointsCount(...)*, *Instantiate(...)* and *Compose(...)* employed by the algorithm. These functions were implemented as ATL transformation models and used to transform several input models to desired target models to achieve composition of SDs. Before giving the implementation details of these transformation models, we would like to first justify some of our design decisions and also describe how we designed our mark model.

5.1 Design decisions

Several key decisions were taken in this work. These include:

- The use of ATL transformation models for composition instead of, say, graph transformations or general programming languages (e.g., Java). Aspect composition or weaving can be considered a form of model transformation because we take at least two input (primary and aspect) models and produce at least one target model (composed). Therefore, model transformation approaches can be used for aspect composition. ATL was chosen because it is mature and has a rich set of development tools that are built on top of flexible and versatile Eclipse platform. ATL is based on OCL; therefore, it is not difficult for a developer with some OCL experience to learn. ATL was also chosen because no work on behavioral aspect composition, that we are aware of, has been attempted using ATL.
- The use of RSA 7.5 as a modeling tool of choice. RSA 7.5 is not free but we already have a license for it. It is a great UML modeling tool. It has excellent support for SDs. It is easy and intuitive to use. It allows for easy model migration. We can export or import UML models as .uml or XMI files. It allows for model validation (not available in some of the tools) which we found very useful. RSA can also generate a sequence diagram from an imported model. This makes verification of our composed model easy and less error prone.
- The use of a mark model. ATL Transformations only work with models; therefore, our binding rules have to be in the form of a model that has a metamodel. Mark models are a convenient way to work with parameterized transformations. MDA, certainly, allows for use of mark models in model transformations (Happe et al., 2009). Happe et al. use mark models to annotate performance models in their work on performance completions (Happe et al., 2009).
- Ignoring BehaviorExecutionSpecifications (BESs) and Execution Occurrence Specifications (EOSs) model elements during pointcut detection and composition. As stated in the previous section, we are convinced that we can ignore these two and still achieve accurate pointcut detection. This is because BESs and EOSs are used to define the duration of execution of behavior (OMG, 2009) of say, a message invocation. Our work is focused on detecting the occurrence of a sequence of messages (interactions between participants) and doing something when we find the sequence. We are not concerned about how long the participant will execute after a message invocation.

During early stages of system modeling or at high levels of system abstraction, BESs and EOSs are not really applicable or useful; therefore, our decision to ignore them is reasonable.

5.2 Designing mark model metamodel

A mark model helps define binding rules for instantiating generic aspect models. These rules are merely template parameter and value pairs stored in mark model instances. In MDE, a model must have a metamodel that it conforms to, and our mark model is no exception. Since the mark model is to be used in ATL transformations (with an EMF model handler), its metamodel must conform to a metametamodel that is the same as the ATL metamodel, that is, Ecore as shown in Figure 4.1 and Figure 4.2. ATL development tools include KM3 (Kernel MetaMetaModel) which is textual notation for defining metamodels (ATL_Manual, 2009). The code snippet below shows a KM3 definition of the metamodel for our mark model which we named *BindingDirectives*. The metamodel has one class with a *parameter* and *binding* value attributes of type *String*. This essentially means that the instances of the mark model will be a collection of objects with initialized parameter and binding attributes.

```
package BindingDirectives {
        class BindingDirective {
                attribute parameter : String;
                attribute binding : String;
        }
        package PrimitiveTypes {
                datatype String;
        }
}
```

The metamodel is defined in a *.km3* file which is then converted (injected) to an Ecore format encoded in XMI 2.0 using injectors in the ATL IDE (ATL_Manual, 2009). Once the metamodel has been defined, we can begin creating mark models in an XMI format.

5.3 Implementation of the complete composition algorithm

As mentioned earlier, the Complete Composition Algorithm uses the *JoinPointsCount*, *Instantiate*, and *Compose* transformations to produce a composed model (SD). These transformations in return implement the other two algorithms (Pointcut detection and Advice Composition algorithm) to achieve their objectives.

5.3.1 Getting the Number of Join Points

The *JoinPointsCount* transformation is implemented by the ATL transformation model shown in Figure 5.1. It returns the number of join points found in the primary model given a pointcut defined in a generic aspect, and binding rules defined in a mark model. The transformation produces a simple UML target model that contains the number of join points found. The number of join points must be returned in a model because an ATL transformation (module) has to produce a model but not a string or integer. The

JoinPointsCount transformation is implemented in an ATL module creatively named **JoinPointsCount** as shown below by its header definition.

```
module JoinPointsCount;
create NUMJOINPOINT:UML2 from PRIMARY:UML2, ASPECT:UML2, BIND:BD;
uses PointcutMatchHelpers;
                                    ...
```

The header declares that the transformation takes as input two UML2 models (bound to variables PRIMARY and ASPECT), a model that conforms to the BD (Binding Directive) metamodel, that is the mark model bound to the variable BIND. The transformation then produces a UML2 target model bound to the variable NUMJOINPOINT. The header also declares that the transformation uses the *PointcutMatchHelpers* ATL library. This is where common or general purpose helpers such as the ones used for pointcut detection (and used also by other transformations) are defined. This helps reduce code duplication and allows for a better code maintenance.

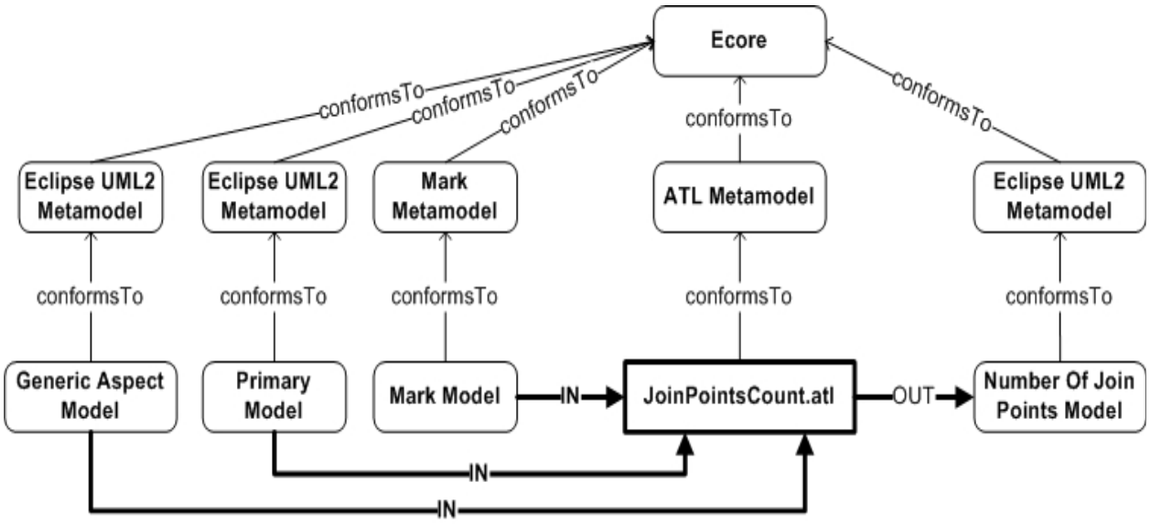


Fig. 5.1. An Overview of the *JoinPointsCount* ATL Transformation.

5.3.2 JoinPointsCount helpers

The transformation employs several helpers listed in Appendix A. It also uses some of the helpers defined in the *PointcutMatchHelpers* library listed in Appendix B. Please note that aspect model here refers to the generic aspect model (not context specific) which is one of the input models to the transformation.

5.3.3 JoinPointsCount rules

Rules are used to generate the target model in ATL. Our *JoinPointsCount* transformation has two simple declarative rules (one matched rule and one lazy rule) that generate a UML model to store the number of join points found. A proper UML model should have a *Model* container element that packages all the other modeling elements. The list of contained objects is then referenced by the *packagedElement* attribute or association. The *Model* matched

rule is the main rule that generates a *Model* element. We want the rule to match only one element. Therefore, its source pattern defines that the rule should match an element of type **UML2 Model** from the input aspect model as it can be seen on line 2 in the code snippet for the rule below. The rule's target pattern defines that a **UML2 Model** element will be generated.

```
1. rule Model {
2.   from s : UML2!Model(s.fromAspectModel() )
3.   to t : UML2!Model (
4.    name <- 'NumberOfJoinpoints',
5.          packagedElement <- Sequence {
6.    thisModule.CreateLiteralInteger(thisModule.numJoinPoints,
7.                    'NumberOfJoinpoints'),
8.                                                    ...
```

The attributes of the target elements will be initialized as defined from line 4. A *Model* element has a name and a collection of packaged elements. The *name* attribute is set to 'NumberOfJoinpoints'. The *packagedElement* attribute will be set to a sequence containing a **UML2 LiteralInteger** element generated by the invoked *CreateLiteralInteger* lazy rule. This lazy rule is passed the number of join points and a string (name) as parameters. The number of join points is, therefore, returned in a **UML2 LiteralInteger** model element packaged in a **UML2 Model** element. The code snippet for the *CreateLiteralInteger* lazy rule is shown below. The rule creates a *LiteralInteger* model element and initializes its name and value attributes with a string (desired name) and an integer (number of join points found by our transformation) respectively.

```
1.    lazy rule CreateLiteralInteger  {
2.      from count : Integer,  name :String
3.      to t: UML2!LiteralInteger  (
4.              name <- name,
5.       value <- count
6.                                                  ...
```

5.4 Instantiating A generic aspect model

The *Instantiate* transformation is implemented by the ATL transformation shown in Figure 5.2. This transformation instantiates a generic aspect model and produce a context specific aspect model. It inputs a primary model, generic aspect model and a mark model, and outputs a context specific aspect model.

The *Instantiate* ATL module, whose header is shown below, implements the *Instantiate* transformation. The header declares that the module creates a target **UML2 model** bound to the CONTEXTSPECIFIC variable.

```
module Instantiate;
create CONTEXTSPECIFIC : UML2 from PRIMARY : UML2, ASPECT : UML2, BIND : BD;
uses PointcutMatchHelpers;
                                                    ...
```

The module has two UML2 source models (bound to variables PRIMARY and ASPECT) and one source model bound to the variable BIND that conforms to our *Binding Directives*

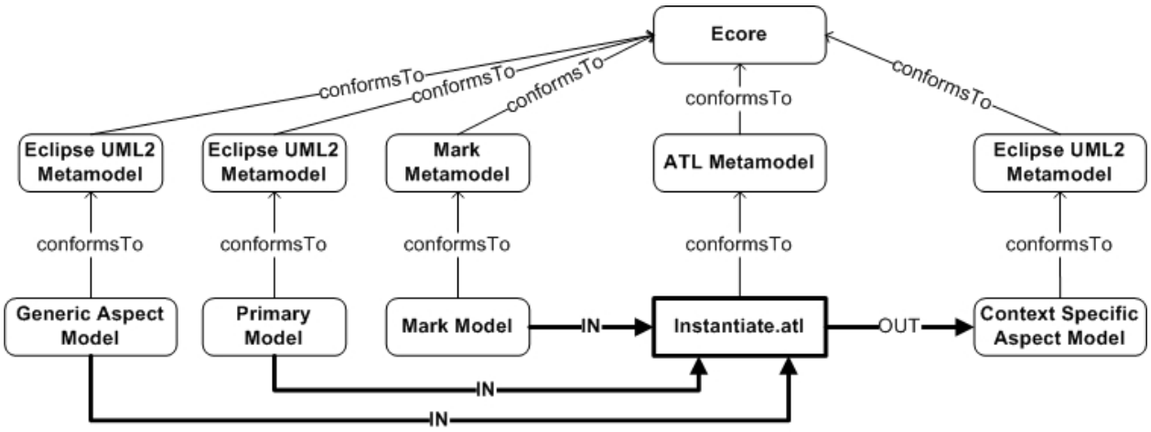


Fig. 5.2. An overview of the *Instantiate* ATL Transformation.

metamodel (BD); that is, a mark model. The header also declares that the module imports the *PointcutMatchHelpers* library.

5.5 Instantiate helpers

Just like the *JoinPointsCount*, this module also uses some of the helpers defined in the *PointcutMatchHelpers* library listed in Appendix B. This transformation also uses helpers defined within its module. Before we can generate the context specific aspect model, we have to ensure that we have a join point where we can weave. The *pointcutMatched* attribute helper returns true if we have at least one join point. It is used as a guard condition for all the rules that generate the target model elements as we shall see later. This ensures that no model element will be generated if there are no join points. The details of this helper are shown below.

```
helper def: pointcutMatched : Boolean =
        thisModule.joinPointsFragments()->notEmpty();
```

The helper returns true if the sequence that contains all the fragments at the beginning of each join point (returned by *joinPointsFragments()*) is not empty. Since *pointcutMatched* is defined as an attribute helper, it is evaluated once and the result cached. This means that successive calls to the helper will be faster which improves performance especially in our case where the helper is called many times by all the rules.

5.5.1 Instantiate rules

Several rules are required to generate a complete context specific aspect model. In fact, we have a rule for every model element type required for a well formed UML sequence diagram. These rules include several matched rules and a handful of lazy rules. Just like in the previous transformation, our target UML model should have a *Model* container element that packages all the other modeling elements. The rule that generates the target *Model* element is shown below.

```
1. rule Model {
2.    from s : UML2!Model (
3.                   s.fromAspectModel() and thisModule.pointcutMatched
4.           )
5.    to t : UML2!Model (
6.           name <- s.createAspectName(),
7.                   packagedElement <- s.packagedElement
8.           )
9. }
```

The source pattern specifies that the rule matches elements of type **UML2 Model**. It also has a condition that the matched element should come from the aspect model (using *fromAspectModel()* helper), and also that *pointcutMatched* must be true, as mentioned earlier. There is only one *Model* element from the aspect model. If at least one join point was found, then only one **UML2 Model** element will be created on the target model since the target pattern declares that the rule creates an instance of **UML2 Model**. Its packaged elements will be initialized to the list from the matched element as defined on line 7 above. The name will be initialized with the string returned by the *createAspectName()* helper on line 6 above. The UML specification describes that an *Interaction* can be contained in a *Collaboration.* Collaborations are used to show the structure of cooperating elements with a particular purpose (OMG, 2009). Indeed, the primary and aspect models created using RSA have interactions contained within collaborations. The *Collaborations* matched rule has the task of generating *Collaboration* objects that enclose the interactions for the advice and pointcut. Recall that the aspect model consists of the advice and pointcut SDs (interactions). The rule is described by the code snippet shown below.

```
1.    rule Collaborations {
2.       from s : UML2!Collaboration (
3.                      thisModule.aCollaborations->includes(s) and
4.                      thisModule.pointcutMatched
5.              )
6.       to t : UML2!Collaboration (
7.              name <- s.name,
8.                      ownedBehavior <- s.ownedBehavior,
9.              ownedAttribute <- s.ownedAttribute,
10.             ownedConnector <- s.ownedConnector
11.             )
12.   }
```

The guard condition for this rule's source pattern ensures that only collaborations from the aspect model (and not from primary model) are matched. It checks if a collaboration is included in the collection of collaborations from the aspect model returned by the *aCollaborations* attribute helper. The attributes of the generated collaboration, including the enclosed interactions (*ownedBehavior*), are initialized from those of the matched collaboration as shown on lines 7 to 10 above. The *aInteractions* and *pInteractions* rules are used to create *Interaction* target elements for the advice and pointcut respectively. The rules are almost identical with slight differences in the source pattern guard. The code below gives details of the *aInteractions* rule. The difference between the rules is in line 3. The guard for *aInteractions*

rule ensures that the rule matches the interaction from the aspect's advice which has the name "Advice".

```
1.  rule aInteractions {
2.    from s : UML2!Interaction (
3.                  s.name = 'Advice' and thisModule.pointcutMatched
4.          )
6.    to t : UML2!Interaction  (
7.     name <- s.name,
8.                  lifeline <- s.lifeline,
9.          fragment <- s.fragment,
10.         message <- s.message,
11.                 ownedAttribute <- s.ownedAttribute,
12.                 ownedConnector <- s.ownedConnector,
13.                 generalOrdering <- s.generalOrdering,
14.         ownedBehavior <- s.ownedBehavior,
15.         covered <- s.covered
16.         )
17. }
```

The guard for the *pInteractions* rule matches the interaction from the aspect's pointcut which has the name "Pointcut". Both rules then initialize the attributes of the generated interactions using the values from the attributes of the matched source elements as it can be seen from lines 7 to line 15.

The *Lifelines* rule generates lifelines for both the advice and pointcut SDs. The rule matches all lifelines from the advice model as shown on line 3 of rule's code snippet on the next page. The *aLifelines* helper returns all lifelines from the generic aspect model (advice and pointcut SDs). The generated lifeline's attributes are then initialized as shown from lines 6 to 8. This rule probably best shows how helpers are used to assist rules in creating the target models other than been used as guard conditions in the source pattern. We can see on line 6 that binding is achieved by using the *bind()* helper to initialize the name of the generated lifeline.

```
1.  rule Lifelines {
2.    from s : UML2!Lifeline (
3.          thisModule.aLifelines->includes(s)and thisModule.pointcutMatched
4.          )
5.    to targetLifeline : UML2!Lifeline (
6.      name <- s.bind(),
7.                  coveredBy <- thisModule.getMOSByLifeline(s),
8.                  represents <- s.represents
9.          )
10. }
```

5.6 Composing aspect models

After obtaining a context specific aspect model from the previous transformation (*Instantiate*), the next step is to compose the context specific aspect model with the primary model. This is achieved by the *Compose* ATL transformation whose overview is shown in

Figure 3.5. The transformation inputs the primary and context specific source models, and produces a composed target model. Both the source models and the output model conform to the UML2 metamodel. This transformation is implemented by the *Compose* ATL module. The code snippet below shows a description of the module's header.

```
module Compose;
create COMPOSED : UML2 from PRIMARY : UML2, ASPECT : UML2;
uses PointcutMatchHelpers;
                                ...
```

As expected, the header declares that the module creates a UML2 Model bound to the variable COMPOSED from two UML2 source models bound to the variables PRIMARY and ASPECT for the primary and aspect models respectively. The module also uses some helpers from the *PointcutMatchHelpers* library.

5.6.1 Compose helpers

Our *Compose* transformation has a number of helpers. All the helpers have a necessary role to play but some roles are, certainly, more important than others. For example, the *getTargetFragments* attribute helper has the privilege of returning the composed sequence of fragments, that is, *sequence {sub_before, A, sub_after}* from algorithm-3. The code definition of this helper is shown below. The helper begins by ensuring that there is, at least, one join point by calling the *pointcutMatched* helper on line 2, which we have described earlier. If there exists a join point, *getTargetFragments* then obtains a sequence of fragments before the join point by invoking the *lowerFragSub* helper on line 4, a sequence of fragments from the advice (by invoking *getAspectFragments*) on line 5, and a sequence fragments after the join point on line 6. It returns a flattened sequence consisting of those sequences. The fragments returned by *getTargetFragments* are used to initialize the fragment attribute of the interaction generated by our transformation.

```
1.  helper def : getTargetFragments : Sequence(UML2!InteractionFragment) =
2.              if thisModule.pointcutMatched then
3.                Sequence {
4.                      thisModule.lowerFragSub(thisModule.firstJPIndex),
5.                      thisModule.getAspectFragments('Advice'),
6.                      thisModule.upperFragSub(thisModule.firstJPIndex +
7.                      thisModule.numPCTFs-1)
8.                    }->flatten()->asSequence()
9.            else
10.                     Sequence{}
11.           endif;
```

The *getTargetFragments* helper also serves as the base of our composition process. Almost all the other elements to be used in generating the target model are rooted from this helper. The *targetCFs* helper, which returns all combined fragments to be used for generating combined fragments in the target model, iterates through fragments returned by *getTargetFragments* returning all instances of *CombinedFragment*. The *targetOperands* helper, in return, iterates through the sequence of combined fragments returned by *targetCFs* to obtain all instances of *InteractionOperand*.

5.6.2 Compose rules

Several rules are defined for creating the composed target model. Rules in the *Compose* transformation probably use more helpers compared to the two previous transformations, mainly because in this transformation more elements are removed or added. This requires modifications to many associations between model elements. The code below is that for the *Model* matched rule which is used to create the **UML2 Model** element.

```
1.  rule Model {
2.    from s : UML2!Model (
3.          s.fromPrimaryModel()  and thisModule.pointcutMatched
4.          )
5.    to t : UML2!Model (
6.                name <- thisModule.getModelName(s.name, thisModule.aModel),
7.                      packagedElement <- Sequence {
8.                      thisModule.targetClasses,
9.                      thisModule.pCollaborations,
10.                           thisModule.getTargetEvents()
11. ...
```

The rule matches elements of type **UML2 Model** from the primary model, and provided the pointcut matches as defined by the source pattern on lines 2 and 3. The rule creates instances of *Model* as declared on line 5. Since the primary model consists of one instance of the *Model* class, this rule will generate only one instance. It then initializes the created instance with the use of several helpers as defined on lines 6 to 11. The *getModelName* helper generates a string used to initialize the name attribute. The *packageElement* list attribute is initialized to a sequence of classes returned by *targetClasses*, a collaboration from the primary returned by *pCollaborations*, and a sequence of events returned by the *getTargetEvents()* operation helper. These three helpers are defined in the context of the module; hence, the use of the keyword *thisModule*.

```
1.  rule Messages{
2.    from s : UML2!Message (
3.          thisModule.targetMessages->includes(s) and
4.          thisModule.pointcutMatched
5.          )
6.    to t : UML2!Message (
7.          name <- s.name,
8.          sendEvent <- s.sendEvent,
9.          receiveEvent <- s.receiveEvent,
10.         messageSort <- s.messageSort,
11.         argument <-  s.argument->collect (e |
12.                     if  e.oclIsTypeOf(UML2!LiteralString) then
13.                           thisModule.CreateLS(e)
14.                     else
15.                           if  e.oclIsTypeOf(UML2!LiteralInteger) then
16.                                 thisModule.CreateLI(e)
17.                           else
18.                                 OclUndefined
19.                           endif
20. ...
```

The *Messages* rule shown below is used to generate messages for the target model. This rule is more interesting since it calls a few lazy rules to help initialize some of the attributes of the target messages to be created. The rule matches all messages included in *targetMessages* and creates messages for the target model. The generated message is initialized as defined from line 7. On line 12, the *argument* attribute is initialized by calling a suitable lazy rule. We are only interested in primitive type message arguments (integers and strings); therefore, we have two lazy rules for creating an instance of *LiteralInteger* or *LiteralString* depending on the argument type for the matched message.

The rule uses ATL's built-in *oclIsTypeOf(t: oclType)* operation to check the type of the argument for the matched message. If it is a *LiteralString* then the *CreateLS* lazy rule is called but if it is a *LiteralInteger* the *CreateLI* lazy rule is called instead. If the argument is neither an integer nor a string, the message's argument attribute is initialized to *OclUndefined*, ATL's equivalent of null. All the rules that are used by the Compose transformation to generate the composed model are listed in Appendix C.

6. Case studies - phone call features as aspects

This case study of a cell phone application was adapted from Whittle and Jayaraman in (Whittle et al., 2007). The application has three use cases but we are only interested in two; namely, *Receive a Call* and *Notify Call Waiting* (Whittle et al., 2007]. The *Receive a Call* use case is considered to be the base model and the *Notify Call Waiting* is considered the aspect. Figure 6.1 shows a dynamic view of the *Receive a Call* use case modeled as a sequence diagram. This will be our primary model. When the user's phone receives a call (*incomingCall* message), it alerts the user by displaying the appropriate information about the caller on the phone's display (Whittle et al., 2007) by sending a *displayCallInfo* message to the display. The phone also sends a *ring* message to the ringer. The user then has several options captured by an **alt** combined fragment. The user can accept the call by sending a *pickUp* message to the phone and later end the call by sending a *hangUp* message. Alternatively, if the user chooses not to accept the call, the user can send a *disconnect* message to the phone. If the user elects to ignore the call, the phone will ring for a specified amount of time and then time out ending the scenario. As mentioned, the *Notify Call Waiting* scenario or feature is considered an aspect. The approach (graph transformations) taken by Whittle and Jayaraman (Whittle & Jayaraman, 2006) does not have the notion of generic or context specific models like our approach. Therefore, Figure 6.2 shows our representation of the behavioral model of the *Notify Call Waiting* scenario as a generic aspect model.

The pointcut is defined as a sequence of parameterized |accept and |end messages from the receiver lifeline to the phone lifeline. This will match a sequence of two messages that will be bound to |accept and |end from the lifeline bound to |receiver. The advice shown in Figure 5.2b is slightly more complex. It introduces messages that if bound properly, will place the current call on hold (Whittle et al., 2007). The behavior defined by the advice is only applicable when the user is currently on call; therefore, we must ensure that the advice is weaved between the *pickUp* and *hangUp* messages on the primary model (Whittle et al., 2007). To achieve this, we will bind |accept and |end to *pickUp* and *hangUp* respectively, as shown on lines 9 and 10 of the mark model below. The |receiver parameter is bound to *user* so the pointcut matches *pickUp* and *hangUp* from the user to the phone.

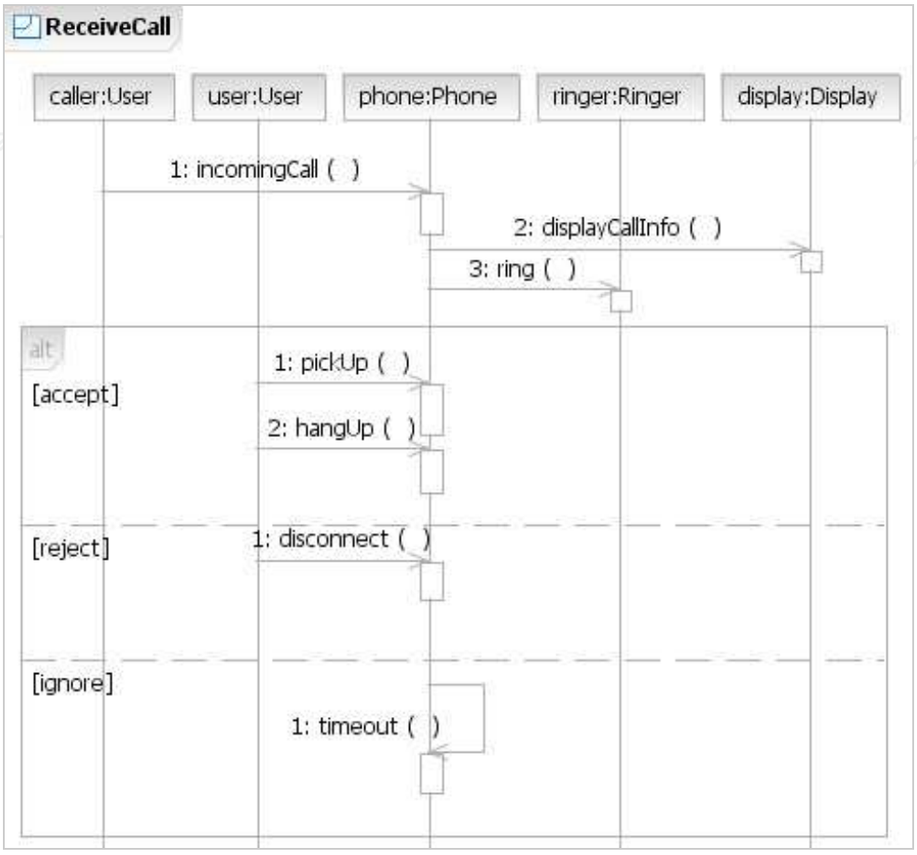
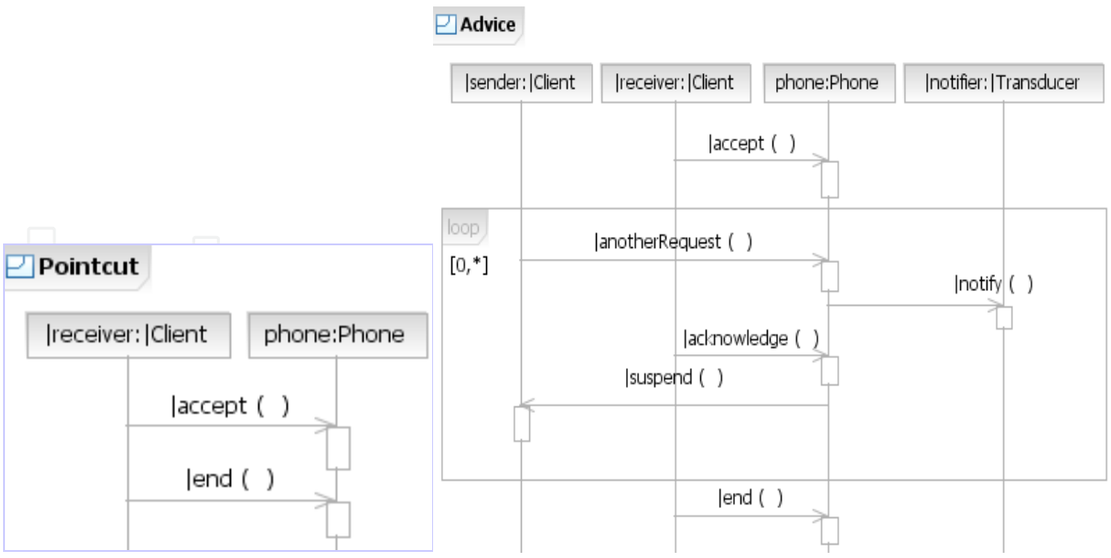


Fig. 6.1. Receive a Call Primary Model Adapted from (Whittle et al., 2007).

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3.  xmlns="BindingDirectives">
4.    <BindingDirective  parameter="|receiver"  binding="user"/>
5.    <BindingDirective  parameter="|sender"    binding="caller"/>
6.    <BindingDirective  parameter="|anotherRequest" binding="incomingCall"/>
7.    <BindingDirective  parameter="|notify" binding="displayCallInfo"/>
8.    <BindingDirective  parameter="|acknowledge" binding="ok"/>
9.    <BindingDirective  parameter="|accept"    binding="pickUp"/>
10.   <BindingDirective  parameter="|end"                 binding="hangUp"/>
11.   <BindingDirective  parameter="|suspend" binding="putOnHold"/>
12.   <BindingDirective  parameter="|Client"    binding="User"/>
13.   <BindingDirective  parameter="|notifier"  binding="display"/>
14.   <BindingDirective  parameter="|Transducer"        binding="Display"/>
15. </xmi:XMI>
```



(a) Pointcut (b) Advice

Fig. 6.2. Notify Call Waiting Generic Aspect Model.

The rest of the parameters are bound as defined by the mark model. Recall that by binding these parameters from the generic aspect model, we are actually instantiating it to produce a context specific aspect model. This is done by the *Instantiate* ATL transformation as discussed in earlier sections. However, before going into the trouble of instantiating a context specific aspect (and composing it with the primary model), we must first determine if the pointcut matches, and if so, how much join points were found. Getting the number of join points is performed by the *JoinPointsCount* transformation which takes the generic aspect, the primary and mark models as input, and produces a target model that contains the number of join points. Executing this transformation produces the model shown in Figure 6.3 (when viewed on Eclipse's uml editor). The model contains a *LiteralInteger* object with the name *NumberOfJoinpoints* and which has a value of one, that is, our primary model has one join point.

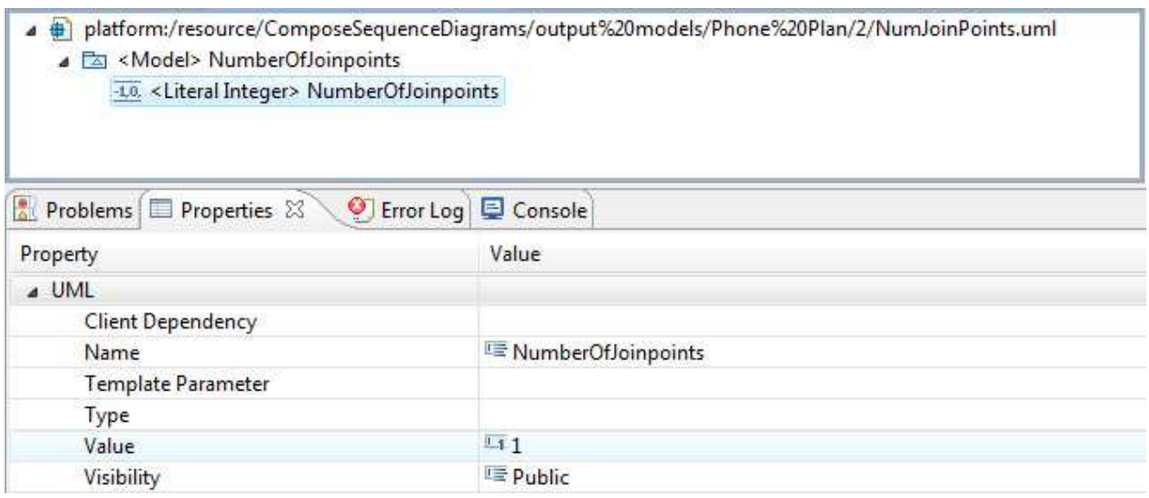
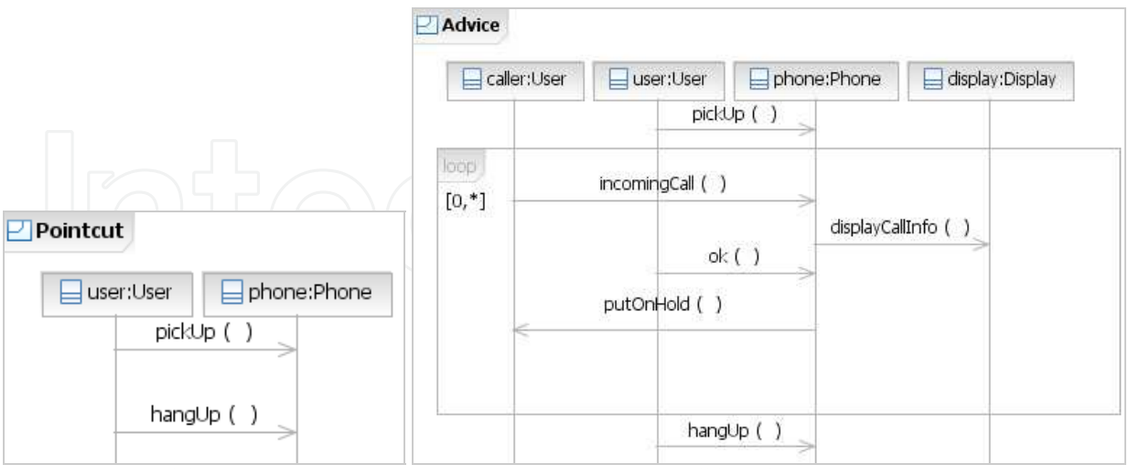


Fig. 6.3. JoinPointsCount Output Model.

With only one join point, our composition algorithm only has to loop once; hence, no loop unrolling is required. The next step is to validate our models using both RSA and our custom *validator* package. Running the *ValidateComposedModel* class from our custom package to validate the composed model (CM) produces the output shown below.


```
Reading model CM from disk ...Validating CM
Model container is valid
All Classes are valid :)
All Class Operations are valid :)
All Message events are valid :)
Collaboration model element ReceiveCall is valid :)
All Owned attributes in ReceiveCall are valid :)
Interaction model element ReceiveCall is valid :)
All Message Occurrence Specifications in ReceiveCall are valid :)
All Messages in ReceiveCall are valid :)
All lifelines in ReceiveCall are valid :)
Our model and all its model elements meet our validation requirements
```



(a) Pointcut (b) Advice

Fig. 6.4. Context Specific Aspect Model.

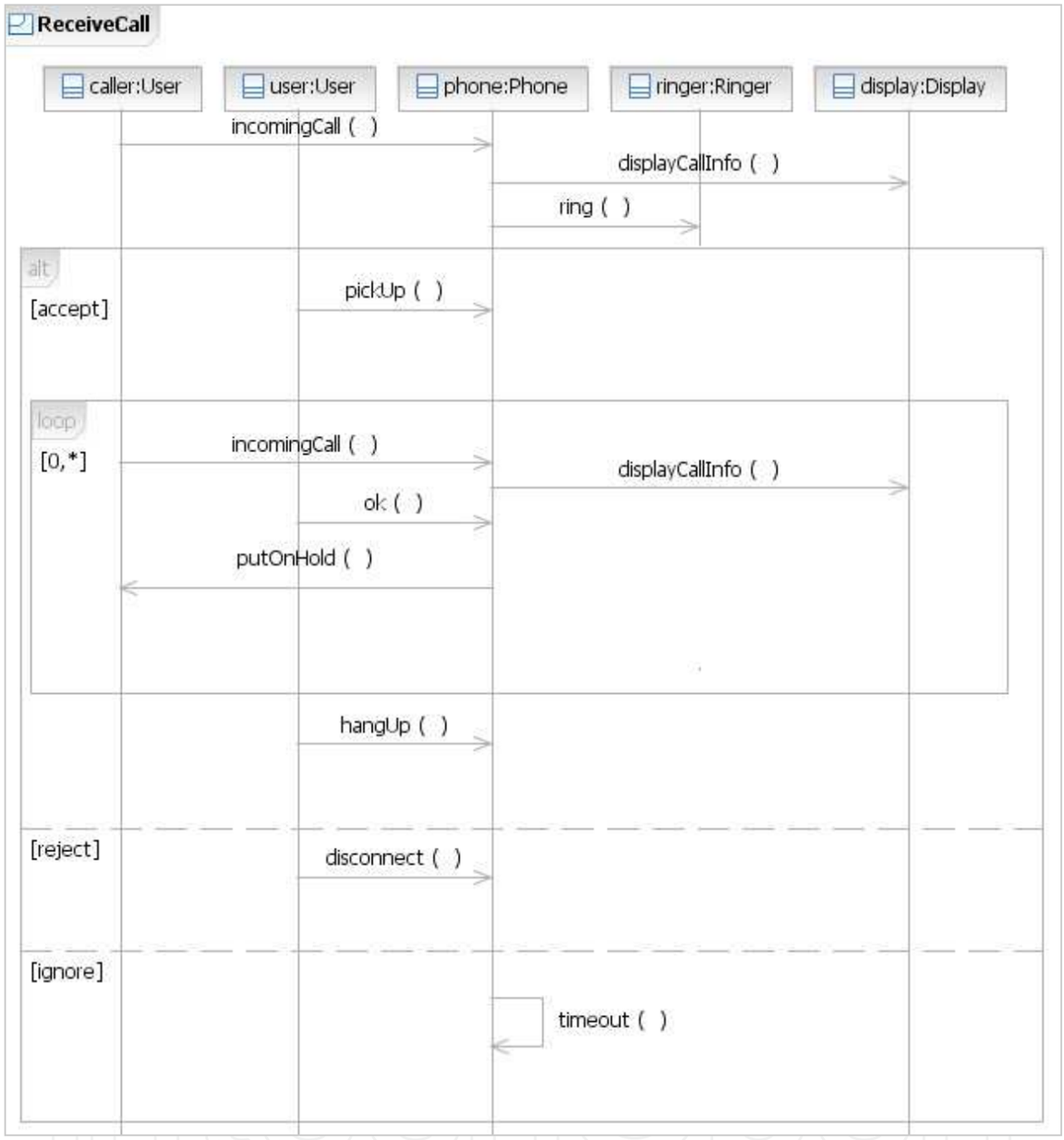


Fig. 6.5. Composed Model.

Running the *ValidateComposedModel* to validate the context specific aspect model (CSAM) also shows that the model is valid. Both models are valid according to the checklist defined in our *validator* package. Both models are then imported into RSA for validation and visual inspection.

Figure 6.4 and 6.5 show the context specific aspect and composed model SDs created on RSA after importing the models. The pointcut was properly bound. We can see in Figure 6.4a that |accept and |end are bound to *pickUp* and *hangUp* while |receiver and |Client have been bound to *user* and *User* respectively. Figure 6.4b shows that the advice has also been bound as specified by the mark model. Figure 6.5 shows a sequence diagram for our

composed model. We can see that the advice has been properly weaved into the **alt** combined fragment's first operand. This way, if the user chooses to accept a call and another *incomingCall* message is received by the phone the phone's display will show the new caller's information. The user can then send an *ok* message to phone to put the caller on hold.

7. Conclusion

The main objective of this work is to compose aspect models represented as UML sequence diagrams (SDs) using the Atlas Transformation Language (ATL). Toward this end, we proposed a formal definition of SDs in terms of an ordered list of interaction fragments, and in the process defined three algorithms for pointcut detection, advice composition and complete composition. We designed and implemented the Complete Composition algorithm to achieve composition of the primary model and generic aspect models. We consider aspect composition as a form of model transformation; therefore, the algorithm is implemented using model transformations written in the ATL model transformation language. We also designed a simple metamodel in Ecore for mark models used to define binding rules which are used to instantiate generic aspect models. We finally designed and implemented a custom Java package to help validate the composed model. The Java classes check the composed model elements against a list of defined constraints designed to ensure that essential model elements are present in the composed model and are properly initialized.

The Complete Composition Algorithm proposed and implemented composes behavioral views of both primary and aspect models represented as UML sequence diagrams. The primary model defines the core system behavior without cross-cutting concerns while the aspect models represent behavior that cross-cuts the primary model. The models are described in UML Sequence diagrams created using an eclipse-based modeling tool (RSA) and then exported to a UML2.1 file for composition and validation.

Using ATL, a mature model transformation language, the Complete Composition Algorithm is implemented using three transformation models; namely JoinPointsCount, Instantiate, and Compose. The JoinPointsCount transformation determines the number of join points in the primary model given the pointcut from the aspect model. The aspect models are made generic so that they can be more reusable; therefore, they must first be instantiated before they can be composed with the primary model. The Instantiate transformation is used to instantiate generic aspect models in the context of the application using a set of binding rules defined in mark models to produce context specific aspect models. The Compose transformation then takes the primary model and context specific aspect model as inputs, and produces a composed model. This process is repeated as many times as there are join points and aspect models until a complete integrated system is obtained.

To test our design and implementation, several test cases and case studies were successfully conducted. Validation was achieved by using custom Java classes to check the model against a set of defined constraints. The composed model was also validated using RSA's built-in validation feature. To verify composition, a sequence diagram was generated from the model's UML2.1 file using RSA. The generated sequence diagram was then visually inspected to see if the composition was performed properly.

Using ATL for composing models does have its challenges. The inability to navigate target models or modify input models makes intricate weaving of aspects a hard problem. However, the benefits of using a versatile language that allows for powerful expressions may outweigh the challenges.

7.1 Limitations and future work

This work is part of an ambitious quest for a complete AOM composition framework and a set of tools that can allow software architects and developers to easily apply AO techniques to model driven software development. There are several limitations that must be addressed, and new features to be added before our composition approach can be more useful. These include:

- Improved string pattern definition and matching for template parameters.
- Non primitive message and operation argument types.
- Support for Interaction Occurrences.
- Support for BehaviorExecutionSpecifications (BESs), ExecutionOccurrenceSpecifications (EOSs) and other model elements.
- Structural view composition.
- Invoking ATL from Java.
- An eclipse plug-in to help in the creation of the mark model.

Our approach currently supports the use of the wildcard "*" for defining template parameters. This gives some flexibility when defining generic aspect models. However, to allow for powerful expressions, we need to use regular expressions. Currently ATL (version 2.0.x) does not support the use of regular expressions for comparing or matching strings. ATL only uses regular expressions for replacing and splitting strings. Future research may include development of a custom string ATL library that will provide helpers that implement regular expression matching operations.

Furthermore, we have assumed that messages in the primary and aspect models have simple arguments that are either strings or integers. However, messages can have arguments that are instances of classes defined in the structural view of the system. Therefore, the current use of lazy rules to create message arguments (and class operation parameters) may not be ideal. Future work would look at a more efficient and elegant way of creating message arguments in the target model. Interaction Occurrences provide a way to reuse and manage complex SDs. They are a notation for copying one SD (basic) into another one, which may be larger (Pilone et al., 2005). Our current composition approach has no support processing interaction occurrences; therefore, future research would look into including interaction occurrences in pointcut detection and composition. This will provide challenges because the use of interaction occurrences means that the primary model SD or the aspect model SDs may contain more than one instance of *Interaction* depending on the number of interaction occurrences.

Recall that BESs and EOSs were ignored in our composition approach. Future work could look into how these model elements can be processed with other interaction fragments. We would also add support for other model elements that are not currently supported; like, connectors, signals (and related events), gates, etc.

Future research would also include composition of other behavioral views (Statechart and Activity diagrams) and structural views (class diagrams) of the primary model and aspect models. Our current approach does achieve some composition of structural model elements from the primary and aspect models but does ignores associations between the structural elements.

8. Acknowledgement

This research work was partly sponsored by NSERC (Natural Sciences and Engineering Research Council) of Canada through grant number EGP 401451-10.

9. Appendix A - JointPointsCount helpers

Helper Name	Return type	Purpose
aMessages	Sequence	Returns all messages from the aspect model.
aOperations	Sequence	Returns all class operations from the aspect model.
aLifelines	Sequence	Returns all lifelines from the aspect model.
aSendEvents	Sequence	Returns all SOEs from the aspect model.
aRecvEvents	Sequence	Returns all ROEs from the aspect model.
aProperties	Sequence	Returns all lifeline properties the aspect model.
allBindings	Sequence	Returns all binding rules from the mark model.

10. Appendix B - PointcutMatchHelpers library

Helper name	Return type	Purpose
adviceMOSEncoding()	String	A tagging string for advice MOSs
getAspectFragments(sd)	Sequence	Returns MOSs and CFs from a given SD from the aspect model.
getPrimaryFragments()	Sequence	Returns MOSs and CFs from a given SD from the primary model.
getFragments()	Sequence	Returns MOSs and CFs within the context CF.
getFragments()	Sequence	Returns MOSs and CFs within the context Interaction Operand.
sameEventType(e1, e2)	Boolean	Returns true if the given *MessageEvents* are both ROE or SOE
PairwiseMatchFragments (src, tgt)	Boolean	Checks if the fragments at the same index from *src* and *tgt* sequences are "equal".

getBinding(name)	String	Retrieves the binding value from the mark model for the given template parameter.
bindingDefined(String)	Boolean	Returns true if a binding value exists for the given parameter.
samePropertyType(pct , core)	Boolean	Returns true if the given *Properties* have the same type (class).
equals(mos) Context = MOS	Boolean	Checks if the context and supplied MOSs are equivalent.
invalidMOSName()	Boolean	Returns true if the context MOS has been tagged meaning it was added from a previous composition iteration
getLifelineMOS()	Sequence	Returns MOSs that cover the context lifeline.
getAspectSD (name)	Interaction	Returns an SD (Advice or Pointcut) from the Aspect Model.
getMessageLifelines (m)	Sequence	Returns the sender and receiver lifelines for the given message.
getSDLifelines (sd)	Sequence	Returns all lifelines in a given SD.
getSDMessages (sd)	Sequence	Returns all messages in a given SD.
createMOSName (code, idx)	String	Generates a name for the context MOS given our tagging string and MOS index in the sequence of MOS.
fromAspectModel() Context = Model	Boolean	Returns true if the context model is from the aspect model.
fromAspectModel() Context = Interaction	Boolean	Returns true if the context Interaction is from the aspect model.
fromPrimaryModel() Context = Model	Boolean	Returns true if the context model is from the primary model.
fromPrimaryModel() Context = Message	Boolean	Returns true if the context message is from the primary model.
notInPrimaryModel() Context = Class	Boolean	Returns true if the context class is from the primary model.
notInPrimaryModel() Context = MessageEvent	Boolean	Returns true if the context MessageEvent is from the primary model.
notInPrimaryModel() Context = Operation	Boolean	Returns true if the context operation is from the primary model.
equals(c) Context = Class	Boolean	Returns true the context class is the same as the supplied class; that is, if they have the same name.
classMatch (c1 , c2)		Returns true if the two classes are equal.

equals(o) Context = Operation	Boolean	Returns true if the context operation is the same as the supplied operation.
equals(e) Context = MessageEvent	Boolean	Returns true if self is the same as the supplied event; that is, they are of the same type and have the same operation.
equals(f) Context = CombinedFragment	Boolean	Returns true if the context CF is the same as the given interaction fragment which also has to be CF.
equals(f) Context = InteractionOperand	Boolean	Returns true if the context interaction operand is equal to the given interaction fragment .
getMOSs() Context = InteractionOperand	Sequence	Returns all MOSs enclosed by the context interaction operand.
getMOSs() Context = CombinedFragment	Sequence	Returns all MOSs enclosed by the context CF.
PairwiseMatchOperands (src,tgt)	Boolean	Returns true if the given sequences of operands have matching pair of operands, that is, the operands at the same index are equal.
PairwiseMatchMOSs (src, tgt)	Boolean	Checks if the elements at the same index from the *src* and *tgt* sequence are equal.
joinPointsFragments()	Sequence	Returns fragments at the start of the join points.
getMaxInt()	ValueSpecificationAction	Returns the *maxInt* value of the given Interaction constraint.
getMinInt()	ValueSpecificationAction	Returns the *minInt* value of the given Interaction constraint.

11. Appendix C – Compose rules

Rule	**Purpose**
Model	Generates a *Model* element that contains all the other target model elements.
Collaborations	Creates a *Collaboration* that contains the composed model interaction.
Interactions	Generates the *Interaction* that owns fragments, lifelines, messages, etc.
CombinedFragments	Generates all *CombinedFragments* including nested ones.
InteractionOperands	Generates all *InteractionOperands*.

InteractionConstraints	Creates all the constraints.
OpaqueExpressions	Generates *OpaqueExpressions* for *InteractionConstraints*.
Lifelines	Generates all lifelines.
Messages	Produces messages for the target model.
MOSs	Creates all *MessageOccurrenceSpecifications* (MOSs).
ROEs	Generates *ReceiveOperationEvents* for *receiveEvent* MOSs.
SOEs	Produces *SendOperationEvents* for *sendEvent* MOSs.
Operations	Generates operations for classes.
Classes	Generates classes.
Properties	Generates all *Properties* for lifelines.
lazy rule CreateLUN	Creates *LiteralUnlimitedNaturals* that are used for guard conditions.
lazy rule CreateLS	Creates strings that are used for guard conditions and arguments for messages.
lazy rule CreateLI	Creates integers that are used for guard conditions and arguments for messages.
lazy rule CreateParam	Creates parameters for operations in the target model.

12. References

http://wiki.eclipse.org/ATL/User_Guide, last accessed on September 22, 2009.

Samuel A. Ajila, Dorina Petriu, and Pantanowitz Motshegwa, *Using Model Transformation Semantics for Aspects Composition*, 2010 IEEE 4th International Conference on Semantic Computing (IEEE-ICSC 2010), pp 325-332, Carnegie Mellon University, Pittsburgh, PA, USA, September 22 - 24, 2010

ATLAS group LINA & INRIA Nantes, "ATL User Manual version 0.7," Online resource available at: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf, last accessed on September 22, 2009.

O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing Multi-View Aspect Models," In Proceedings of the Seventh International Conference on Composition-Based Software Systems, pages 43-52, 2008.

A. Colyer, A. Clement, G. Harley, M. Webster, "Eclipse AspectJ. Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools,"Addison Wesley Professional, December 14, 2004, ISBN : 0-321-24587-3

M. Didonet Del Fabro, J. Bézivin, and P. Valduriez, "Weaving Models with the Eclipse AMW plug-in," In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany, 2006.

F. Fleurey, B. Baudry, R. France, and S. Ghosh, "A Generic Approach for Automatic Model Composition," Lecture Notes In Computer Science archive Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007,pages 7–15, 2008, Springer-Verlag Berlin, Heidelberg.

R. France, I. Ray, G. Georg and S. Ghosh, "Aspect-Oriented Approach to Design Modeling," IEEE Proceedings – Software, Special Issue on Early Aspects: Aspect –Oriented Requirements Engineering and Architecture Design, 151(4):173-185, August 2004.

H. Gong, "Composition of Aspects Represented as UML Activity Diagrams.", Master's thesis, Carleton University, 2008.

I. Groher, and M. Voelter. "XWeave: models and aspects in concert," In Proceedings of the 10th international workshop on Aspect-oriented modeling, March 2007.

K. Hamilton, and R. Miles, "Learning UML 2.0", O'Reilly, April 2006, ISBN-10: 0-596-00982-8 .

J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric Performance Completions for Model-Driven Performance Prediction," Journal of Systems and Software, Volume 82 Issue 1, January 2009, Elsevier Science Inc.

I. Jacobson, and P. Ng, "Aspect-oriented software development with use case," Addison-Wesley Professional, December 30, 2004, ISBN-10: 0321268881.

C. Jeanneret, R. France, and B. Baudry, "A reference process for model composition," In Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling, April 2008.

J.M. Jézéquel, "Model Transformation Techniques Model Techniques," Online resource available at: http://modelware.inria.fr/static_pages/slides/ModelTransfo.pdf, last accessed in August 30, 2009

F. Jouault, and I. Kurtev, "Transforming Models with ATL," In proceedings of the Model Transformation in Practice Workshop, October 3rd 2005.

J. Kienzle, W. A. Abed, and J. Klein, "Aspect-oriented multi-view modeling," In Proceedings of the 8th ACM international conference on Aspect-oriented software development, March 2009.

J. Klein, L. Hélouët, and J.M. Jézéquel, "Semantic-based Weaving of Scenarios," AOSD 06, March 20-24, Bonn, Germany, 2006

J. Klein, F. Fleurey, and J.M. Jézéquel, "Weaving Multiple Aspects in Sequence Diagrams," Transactions on AOSD III, LNCS 4620, pp. 167–199, 2007

Kompose website, http://www.kermeta.org/kompose/ last accessed on September 23, 2009.

M. Milanovic, "Complete ATL Bundle for launching ATL transformations programmatically." Online resource available at: http://milan.milanovic.org/download/atl.zip, last accessed on September 10, 2009.

B. Morin, J. Klein, O. Barais, and J.M. Jézéquel, "A generic weaver for supporting product lines," In proceedings of the 13th international workshop on Software architectures and mobility, Leipzig, Germany, May 2008.

Object Management Group, "UML Superstructure Specification, v2.1.1," Online resource available at: http://www.omg.org/docs/formal/07-02-05.pdf, last accessed last accessed on July 20, 2009.

Object Management Group, "UML Superstructure Specification, v2.2," Online resource available at: http://www.omg.org/docs/formal/09-02-02.pdf, last accessed on August 22, 2009.

OpenArchitectureware website, http://www.openarchitectureware.org/ last accessed on September 23, 2009.

D.C. Petriu, H. Shen, and A. Sabetta, "Performance Analysis of Aspect- Oriented UML Models", Software and Systems Modeling (SoSyM), Vol. 6, No. 4., pages. 453-471, 2007, Springer Berlin / Heidelberg

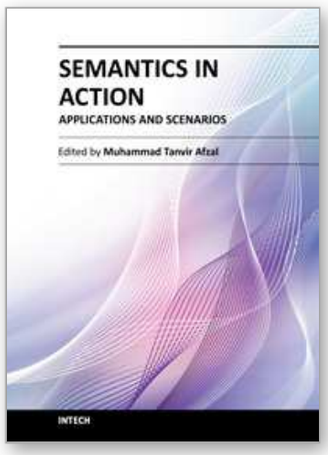
D. Pilone, and N. Pitman, "UML 2.0 in a Nutshell", O'Reilly, (June 2005) ISBN: 0-596-00795-7

G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman, "Model Composition Directives," LNCS 3273, pages 84-97, 2004, Springer Berlin / Heidelberg

J. Whittle, J. Araújo, and A. Moreira, "Composing aspect models with graph transformations," In Proceedings of the 2006 international workshop on Early aspects at ICSE, pages 59-65, Shanghai, China, 2006

J. Whittle, and P. Jayaraman, "MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation," At MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling, Nashville TN USA, Oct 2007.

M. Woodside, D.C. Petriu, D.B. Petriu, J. Xu, T. Israr, G. Georg, R. France, J.M. Bieman, S.H. Houmb, and J. Jürjens, "Performance analysis of security aspects by weaving scenarios extracted from UML models," Journal of Systems and Software, Volume 82 , Issue 1 (January 2009), pp 56-74, 2009



Semantics in Action - Applications and Scenarios
Edited by Dr. Muhammad Tanvir Afzal

ISBN 978-953-51-0536-7
Hard cover, 266 pages
Publisher InTech
Published online 25, April, 2012
Published in print edition April, 2012

The current book is a combination of number of great ideas, applications, case studies, and practical systems in the domain of Semantics. The book has been divided into two volumes. The current one is the second volume which highlights the state-of-the-art application areas in the domain of Semantics. This volume has been divided into four sections and ten chapters. The sections include: 1) Software Engineering, 2) Applications: Semantic Cache, E-Health, Sport Video Browsing, and Power Grids, 3) Visualization, and 4) Natural Language Disambiguation. Authors across the World have contributed to debate on state-of-the-art systems, theories, models, applications areas, case studies in the domain of Semantics. Furthermore, authors have proposed new approaches to solve real life problems ranging from e-Health to power grids, video browsing to program semantics, semantic cache systems to natural language disambiguation, and public debate to software engineering.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Samuel A. Ajila, Dorina Petriu and Pantanowitz Motshegwa (2012). Using Model Transformation Language Semantics for Aspects Composition, Semantics in Action - Applications and Scenarios, Dr. Muhammad Tanvir Afzal (Ed.), ISBN: 978-953-51-0536-7, InTech, Available from: http://www.intechopen.com/books/semantics-in-action-applications-and-scenarios/using-model-transformation-language-semantics-for-aspect-composition

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen